



EE Expert Robert Ashby

Embedded Engineering

[Archive](#) | [Main EE Expert](#) | [Guides & Experts](#)**Integer Square Roots**by [Robert Ashby](#)

A while back, I wrote the article [Methods: Multiplication and Division Made Easy](#). Some of the advantages to the methods that I described were a small use of RAM, speed, and you could expect how long the process would take, i.e., dividing 16/2 or 16000/2 would never take more loops than the bits in your original numbers.

I have queried my co-workers often enough to find an easy way to calculate a square root that didn't involve the obvious loop described below.

```
square_root(number)
  x=1
  while (x^2 < number)
    x++
  return(x)
```

There are obvious shortcomings to this approach that are similar to the problems we overcame with multiplication and division. You don't know how long the process is going to take and it involves a multiplication every loop.

Well I finally came across a good article, [Integer Square Roots](#). Jack Crenshaw wrote it in **Embedded Systems Programming**. He actually attributes the method to a co-worker that he had many years earlier. I salute you both. It was a very informative article. I will try to catch the method in a much shorter form here, but I recommend that you read the original article.

Let's look at some basic algebra.

$$(n + 1)^2 = n^2 + 2n + 1$$

Note: I kind of find the square of the next integer by adding $2n + 1$ to the current square. Also note that $2n + 1$, as $n = 1, 2, 3, \dots$ is simply the odd integer sequence 1, 3, 5, ...

Let's rewrite the above equation.

$$n^2 + 2n + 1 = n^2 + n + (n + 1)$$

This yields the following table.

Initial Number	Numbers Added	Result
0	0 + 1	1
1	1 + 2	4
4	2 + 3	9
9	3 + 4	16
16	4 + 5	25
25	5 + 6	36

Pretty cool, the numbers in the **Initial Number** column are the squares of the left number in the **Numbers Added** column, and the numbers in the **Result** column are the squares of the right numbers in the **Numbers Added** column. I little thought that you could easily put that equation into a loop and compute squares without performing a lot of overhead and with simple addition. The link though is that you can use this same formula backwards and find the closest square root integer. Observe.

Initial Number	Numbers Subtracted	Result
36	0 + 1	35
35	1 + 2	32
32	2 + 3	27
27	3 + 4	20
20	4 + 5	11
11	5 + 6	0

If you go negative, you know that you have exceeded the number you are looking for. This example works out well, since the square of 36 is an integer.

Jack makes the point that multiplying (squaring) any single digit will produce, at the most, a two-digit result. If I understand math correctly, multiplying any two-digit numbers in any base will not give a product that exceeds (in digits) the sum of digits of the original two numbers, e.g., $99 * 9 = 891$, or three digits when I started with a two-digit and a one-digit number. This concept may seem simple, but it is a good thing to remember when in the embedded world. We have to allocate memory for the expected result, but don't want to waste valuable RAM when it's not needed. The other great advantage to this concept is that squaring a single digit doesn't have an effect on the hundreds digit (disregard carries). Let's see what that allows us to do.

Let's take a large number (Jack uses 1234567890). We'll take 123456. What you do is separate the number into sets of numbers, each containing two digits. You then implement the subtract method to find the square of each digit. As you proceed from left to right, make sure that you carry over any remainder from the previous two digits. The number that you start with for each column is the square root of the previous column * 10.

12			34			56		
12	-0 - 1	11						
11	-1 - 2	8						
8	-2 - 3	3						
		3	334					
			334	-30 - 31	273			
			273	-31 - 32	210			
			210	-32 - 33	145			
			145	-33 - 34	78			

			78	-34 - 35	9			
					9	956		
						956	-350 - 351	255

Surprise, Surprise! The integer section of the square root of 123456 is 351. Congratulations. Now, how do we implement this all together? Jack shows some C-type code that is quite simple. I'm going to take a different approach that matches our table here. The following example uses Zilog assembly and some general-purpose registers.

```

;r0 holds the byte that I want the square root of
;r1 holds the remainder
;r2 holds the square root answer
;r5 is a counter
;r7 is a temporary scratch register

square_root:
    clr    r1
    clr    r2
    ld     r5,#4
;set up for 8 bits (Remember we are grouping twos)
loop:
    add    r2,r2
;result is shifted by a digit (no effect first time through)
;shift two highest bits of number into remainder
;we aren't concerned about what gets shifted into the bottom
;of the original number
    rlc    r0
    rlc    r1
    rlc    r0
    rlc    r1
;add the 2n+1 to get the next square
    ld     r7,r2    ;copy result
    add    r7,r7
    add    r7,#1    ;the r7 contains 2n+1
    cp     r1,r7    ;compare to remainder
    jr     ult,next  ;if not able to subtract continue
    sub    r1,r7    ;if able to subtract, do so
    inc    r2       ;increment result
next:
    dec    r5       ;check for the end
    jr     nz,loop
    ret

```

There are some optimizations that could be added to that code, but I tried to leave it looking as much like the tables above as possible. Now you have an idea how to implement a square root routine in a small micro. Go amaze your co-workers! (You're friends probably won't get it.) Thanks, Jack!

Embedded Engineering

[Guides and Experts](#) [Analog Avenue](#) [PLD EDA Tools](#) [PLD](#) [DSP](#) [EDA](#) [Embedded Systems](#) [Power](#) [Test](#)

Copyright ©1999 ChipCenter
[About ChipCenter](#) ■ [Contact Us](#) ■ [Hot Jobs at ChipCenter](#) ■ [Privacy Statement](#) ■ [Advertising Information](#)